Toward A Methodology For
Platform-Based Design of Embedded Systems

By
Bob Altizer
BASYS Consulting
bob.altizer@basysconsulting.com

ABSTRACT - *Business groups have frequently stated that product time-to-market is too long, requirements are difficult to determine and pin down, and development is too costly. A product family oriented approach to platform-based design of embedded systems can ameliorate many of these problems, and such an approach is generally applicable at many levels of product design.*

KEY WORDS – *platform, platform-based design (PBD), embedded systems, SOC, product families, architecture, economic model*

Background

The notion of platform-based design (PBD) is widely believed to be the "next big thing" in embedded system and system-on-chip (SOC) based product development. But presently the SOC industry has no standards for defining or creating formal engineering specifications for a reusable hardware-software platform, nor a standard set of automated design and integration tools to use.

Traditionally many SOC companies have used a *platform product* approach, building an initial product for a market or application domain, then deconstructing and rebuilding it to create later derivatives. SOC companies are also big users of bottom-up design, where well-understood hardware components are aggregated together into subsystems supporting specific products. This paper describes an alternative process for defining and scoping scaleable, reusable families of products and defining platforms on which they are based.

In December, 2001 the Virtual Socket Interface Alliance (VSIA, http://vsi.org) established a Platform-Based Design Study Group, co-chaired by the author of this paper. With members representing several SOC industry leaders (Alcatel, ARM, Cadence Design Systems, Infineon, Mentor Graphics, Motorola, Nokia, Palmchip, Philips, ST Microelectronics, Synopsys, and Toshiba), the study group's charter is to define the basic issues in PBD for deeper analysis by a subsequent development working group (DWG), and to ensure architectural consistency of its work with other VSIA DWGs working on system-level design, functional verification, and hardware-dependent software.

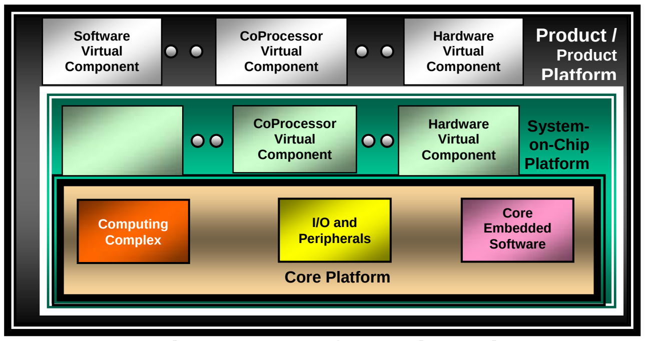

Software Virtual Component
CoProcessor Virtual Component
Hardware Virtual Component
Product / Product Platform
CoProcessor Virtual Component
Hardware Virtual Component
System-on-Chip Platform
Computing Complex
I/O and Peripherals
Core Embedded Software
Core Platform

Figure 1: Platform Hierarchy

Platform-based design usually involves a series of *integration platforms* [1] at multiple levels of abstraction within a product. Selected components at a given layer are integrated into a single, atomic entity—the platform—which is integrated with other functional components to form a higher-level system. Treating platforms as reusable black boxes is an important tenet in PBD; they're "a capability below which you don't need to go to use it." [2]

Figure 1 shows a hierarchy of core, SOC, and product platforms. *Core platforms* are by definition the most fundamental: there is no lower level platform in the hierarchy. They contain a computing complex (general purpose CPUs, control processors, DSPs, etc.), elementary I/O, peripheral, and control functions, and embedded software, such as an operating system kernel. SOC designers can combine core platforms with other market or domain-specific on-chip hardware and software Virtual Components (VCs) to create *SOC platforms*. System or product designers will integrate SOC platforms with their domain specific off-chip functionality to produce products or *product platforms*, which may be integrated at yet higher levels. Appropriate integration-oriented design flows, collateral intellectual property (IP), tools, and support are required at each level for the integration to succeed.

Benefits of Platform Based Design

Platform based design drives the rapid development of derivative products through systematic reuse of validated components, resulting in dramatic reductions in time-to-market and improvements in margin. The basic premise applies regardless of the integration level: the designer who incorporates a platform into a higher-level system or product will have the assurance of using a fully verified and characterized sub-system.

We'll also use platforms to plan and *leverage product families* (groups of products sharing a common platform and managed set of features, that satisfy needs of a selected market), reduce time between successive products in a product family, and drive the systematic development and reuse of IP used to generate the multiple products that constitute the family.

Some of the other benefits of platform-based design include:

- Interface-based design (focusing on defined buses and software services and APIs) promotes higher-level abstraction design and implementation
- Planned design reuse yields very high productivity, and more agility in meeting customer needs
- Designs can be composed of diverse, specialized functions from multiple sources, including externally licensed IP
- Modular design promotes hierarchical on-chip routing and timing, reducing design complexity
- Common strategies are used for multi-processor system design, debug and test using standardized interfaces
- Platform-based derivative products require less verification, and thus have higher margins
- Multiple reuse for blocks allows for development cost amortization and more optimal block design
- Product and platform roadmapping become effective tools in anticipating customer needs
- Effort can be leveraged across multiple projects across the sector, not just one at a time
- Less time spent reacting to customer emergencies means more time for working on needed features

Studies of platform-based design [3] have pointed out the value of separating the actual platform and product design activities, to the point of having separate development groups within a large company do each activity. In this case platform and product development would be loosely coupled, using the so-called "dual life cycle" approach. (Note this is consistent with the general view that platforms, and the VCs with which they're combined to form higher level systems, should in principle be available from a reuse repository or catalog, and eventually licensed freely between creators and consumers throughout an industry segment.)

We can further generalize the dual life cycle approach by starting with a top-down look at the needs of product families, rather than a bottom-up look at platform construction. A product family is just a set of related products in the market; individual member products may be available simultaneously (i.e., at different price/performance points) or a single product may have successive versions come out over time (e.g., Adobe Photoshop). In fact, it's common to have a set of products in the market simultaneously, and with the whole set succeeded by a new model year (examples include everything from consumer electronics to automobiles) .

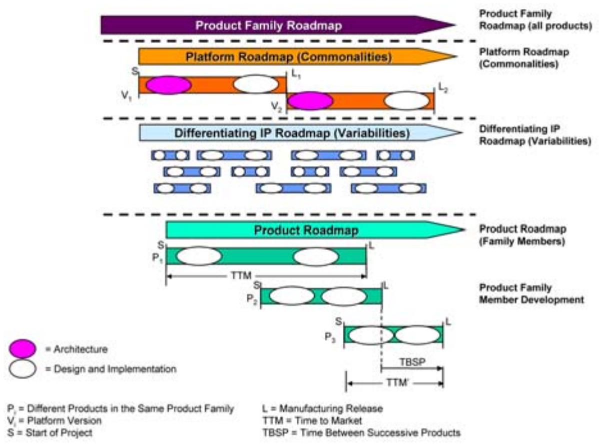

Product Family Roadmap
Product Family Roadmap (all products)
Platform Roadmap (Commonalities)
Platform Roadmap (Commonalities)
S
L1
V1
V2
L2
Differentiating IP Roadmap (Variabilities)
Differentiating IP Roadmap (Variabilities)
Product Roadmap
Product Roadmap (Family Members)
S
L
P1
TTM
S
L
P2
Product Family Member Development
S
L
P3
TBSP
TTM'
= Architecture
= Design and Implementation
Pi = Different Products in the Same Product Family
Vi = Platform Version
S = Start of Project
L = Manufacturing Release
TTM = Time to Market
TBSP = Time Between Successive Products

Figure 2: The Product Family Approach

Product families are a natural application for platform based design. All members of a product family share some common features, while each individual family member will have its own differentiating features. Learning how to manage the commonality and variability of features in a product family—that is, which common features are aggregated into the platform and which remain distinct—can be a big advantage. Figure 2 shows a view of the *product family approach*, in which analysis of the product family roadmap drives separate development of platforms (commonalities) and differentiating IP (variabilities), and product development is primarily an integration process.

The product family approach is clearly applicable in general to any kind of product development. In the balance of this paper we'll see how to apply it to SOC-based products.

Defining Platforms

The first problem you encounter when beginning a study of platform-based design is that there is no standard description of just what constitutes a platform. Even in the limited scope of the SOC and embedded systems industry, working definitions abound, but nothing is definitive. Everyone has their own favorite description.

In December, 2001 the Virtual Socket Interface Alliance (VSIA, http://vsi.org) established a Platform-Based Design Study Group, co-chaired by the author of this paper. With members representing several SOC industry leaders (Alcatel, ARM, Cadence Design Systems, Infineon, Mentor Graphics, Motorola, Nokia, Palmchip, Philips, ST Microelectronics, Synopsys, and Toshiba), the study group's charter is to define the basic issues in PBD for deeper analysis by a subsequent development working group (DWG), and to ensure architectural consistency of its work with other VSIA DWGs working on system-level design, functional verification, and hardware-dependent software. From study group members' point-of-view presentations [2] it's clear that while everyone sees the importance and potential value in developing a PBD capability, no company or individual has a satisfactory answer to the question, "OK, so just what *is* a platform?"

Though these and other groups agree with the OPEN project's basic notion that platforms are integrated aggregations of IP and should serve as the basis for derivative products, it's still difficult to answer many important questions:

- What SOC and core platforms will SPS actually build, and how will we support them?
- What products will be built on which core platforms? On which SOC platforms?
- How will platforms evolve over the lifetime of the product family they support?
- How can platforms be reused across several product lines or product families?
- What are the architectural description standards for core and SOC platforms?
- How are RTOS and other software functions partitioned between the core and SOC platforms?
- What are the details of the interfaces between core/SOC and SOC/product platforms?
- How do reconfigurable processors fit into core platforms?
- What are the test and development tool interfaces at each platform level?

It's Not As Simple As A-B-C

Many organizations approach platform development in an empirical manner you might call the "ABC Approach": Ad hoc, Bottom-up, and Core-centric. The platform specification process in this approach focuses on specific applications built around specific computing cores. Platform architectures are usually the product of lengthy and difficult negotiations between platform developers and their customers in product development groups, attempting to define platform templates based on the customers' functionality needs for just one or two specific SOC products. Customers insist on platforms highly optimized to their own domain needs: they aren't interested in costs (extra functionality, die space, power usage, etc.) that won't benefit their product. This results in narrowly defined subsystems, not flexible and evolvable platforms.

The OPEN project is based within SPS ASP, the organization that will have primary responsibility for core platform development and delivery. Early in the OPEN project we interviewed over 30 stakeholders to capture their concerns and attitudes about platform-based design. These data helped us define the purpose and mission of core platforms; the appropriateness of core platforms for use in solving system problems and improving time-to-market; the feasibility of constructing core platforms; and the stakeholders' views of the risks, integrability, maintainability, deployability, scalability, and evolvability of core platforms [4].

In general (with some vocal exceptions!), stakeholders said that using a platform-based approach is a good idea, and some even indicated that the increasing importance of time-to-market was beginning to override their desire for full custom solutions. But they also had many concerns. First, the tactical issue of delivering to meet commitments: Groups that would develop and deliver platforms are frequently strapped for time and often behind schedule in providing basic core designs themselves, not to mention more elaborate integrated core-based platforms.

The stakeholders' next major concern is domain specificity (also known as controlling of one's own destiny). Though some were willing to trade off design optimization for time to market benefits, none wanted to forego optimization entirely. The notion of "good enough" optimization through use of domain-specific platforms has growng acceptance. Domain-specific platforms are likely a necessity; as no product developer wants to be constrained to a "one size fits all" generic platform. In addition, stakeholders to have widely diverse domain requirements, from extremely low cost, high volume, short lifetime products in the wireless marketplace to the arduous environmental and long life cycle requirements of the automotive market.

Finally, the stakeholders made it clear that for platform-based design to succeed, the full hardware and software tool set, including design and programming tools and a seamless design flow, must be ready along with platforms and integrateable IP components.

The Platform Postulates

In the course of our discussions with stakeholders we developed a set of Platform Postulates, basic statements about the nature of platform-based design we could all agree upon and help foster understanding of issues surrounding platform-based design.

1. Platforms are aggregations of functional hardware and software components ready for integration with other components to form higher-level systems or derivative products.
2. Many levels of integration platforms may exist within a given product or system architecture. In the SOC domain, c*ore platforms* are the most fundamental level.
3. Successful platform-based development depends on the existence not only of defined and verified domain-specific platforms, but also integrateable IP components, an integration-oriented design flow, and tool, application and systems support.
4. Economic advantage from a platform-based product family will be realized through systematic reuse, enhanced product capability (both functionality and flexibility), improved time to market, improved margins, and improved quality.
5. Economic advantages will be large enough to justify investment in development of platforms, integrateable components, collateral IP, tools, and support.

We believe each of these assertions to be true in an effective platform-based design environment. As we'll see later, this list help drive our interest in, and ultimate selection of, the product family oriented approach to platform specification.

The Template Approach

Before getting to the approach we've chosen to employ, let's look at the best of the current bottom-up methodologies. One of its key elements is identification of a basic set, or *template*, of platform functions, which can be used to specify the characteristics of a platform to be used in a certain application domain. In the *template approach*, platform and product developers must agree on the definition and characteristics of each functional component contained in the platform, such as those shown in Table 1. Each of the core platform functional components represents a certain function or set of functions, composed of one or more reusable, pre-verified hardware blocks and/or software modules, which may be static or reconfigurable during platform startup. (Note that the computing complex function represents the computing unit(s) upon which the core platform is based, ranging from a single standard processor core to multiple cores and special processors in homogeneous or heterogeneous configurations.)

Table 1: Typical Core Platform Components

• Computing Complex	• Memory	• SCI, SPI, CAN	• Ethernet
• MMU	• Interrupt control	• DMA control	• USB
• Cache	• Peripheral control	• GPIO	• Firewire
• FPU	• External bus interface	• Timers	• I^{2}C
• MAC	• Master/slave arbitration	• LCD control	• Bluetooth

Designers can successfully use the template approach to define subsystems of specific application domain products, which may work quite well and realize a degree of optimization approaching a full-custom solution. But it due to its reliance on merging a bottom-up (or processor-centric) design with product requirements the template approach has some inherent shortcomings:

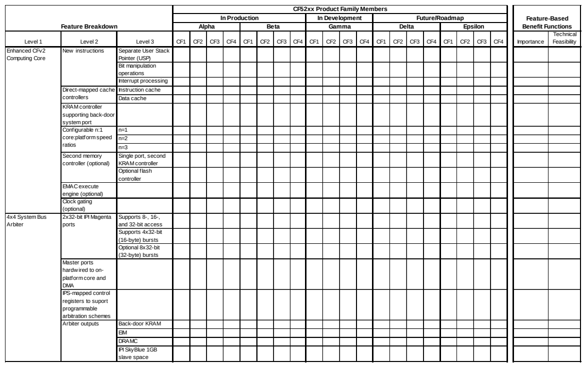
- Because each template-based platform is an empirical design for a particular set of product requirements, its evolution (in feature set support) and migration (to future product generations) is limited.
- Because the template-based platform is typically hardware or processor-centric, it does not involve system software or collateral considerations early in its design phase.
- Because its requirements are defined at the lowest design levels, rather than at the product feature level, template-based platforms may not be easily incorporated into new or different product families other than their original.

The Product Family Approach

Research in domain analysis [3] has shown the bottom-up approach to platform specification to be exceedingly difficult due to its dependence on low-level design detail and (usually) ill-defined requirements, as well as the inherent complexity of beginning design at the detailed, rather than abstract, level. In our experience, incompleteness and instability of customer requirements are frequently reported as significant project problems.

The more generalized *product family approach* [4, 5] is quite powerful in simplifying the platform architecture problem. It starts from the point of view of the functional and performance needs of all the product family members, and then decomposes those requirements to determine which detailed features the members have in common. The common features are analyzed to determine which make most sense—from technical, market, and economic perspectives—to build as part of a platform. In essence, the platform architecture is derived from the product family requirements specification. Those features identified as variable or differentiating should be built as separate IP functional components outside the platform, and integrated with the platform to build specific product family members.

The primary tool of the product family analysis is the *product map*, a detailed spreadsheet or database that captures all the feature, benefit, cost, and value information about the entire product family based on the its members' specific feature requirements. The product map is based on a complete overview of the product family, and can be built at any point in the product family life cycle, including information from products already built and in operation, products under development, and future products still on the roadmap. While the product map looks superficially like the template used in the bottom-up approach, it is the result of significantly more structured analysis, including development and use of characterization and benefit functions that help capture all of the rationale behind its construction. In addition to functional components identified in the template approach, the product map addresses non-functional, collateral, and system-level aspects of the platform. Using the product map, platform developers can perform what-if economic scoping analyses to determine optimum configurations of platform and differentiating features. An example of a product map in use on the OPEN project pilot is shown in Figure 3.



Feature Breakdown			CF52xx Product Family Members																				Feature-Based Benefit Functions	
			In Production								In Development				Future/Roadmap									
			Alpha				Beta				Gamma				Delta				Epsilon					
Level 1	Level 2	Level 3	CF1	CF2	CF3	CF4	CF1	CF2	CF3	CF4	CF1	CF2	CF3	CF4	CF1	CF2	CF3	CF4	CF1	CF2	CF3	CF4	Importance	Technical Feasibility
Enhanced CFv2 Computing Core	New instructions	Separate User Stack Pointer (USP)																						
		Bit manipulation operations																						
		Interrupt processing																						
	Direct-mapped cache controllers	Instruction cache																						
		Data cache																						
	KRAM controller supporting back-door system port																							
	Configurable n:1 core:platform speed ratios	n=1																						
		n=2																						
		n=3																						
	Second memory controller (optional)	Single port, second KRAM controller																						
		Optional flash controller																						
	EMAC execute engine (optional)																							
	Clock gating (optional)																							
4x4 System Bus Arbiter	2x32-bit IPI Magenta ports	Supports 8-, 16-, and 32-bit access																						
		Supports 4x32-bit (16-byte) bursts																						
		Optional 8x32-bit (32-byte) bursts																						
	Master ports hardwired to on-platform core and DMA																							
	IPS-mapped control registers to suport programmable arbitration schemes																							
	Arbiter outputs	Back-door KRAM																						
		EIM																						
		DRAMC																						
		IPI SkyBlue 1GB slave space																						

Figure 3: Product Map Example

Figure 3: Product Map

Central to creation and evaluation of the product map are top-level business and engineering goals for the overall product family, elicited from all relevant stakeholders: marketing, key customers, finance, engineering, manufacturing, channel sales, and so on. Product family goals can be identified and refined using the well known Goal-Question-Metric (GQM) process [5].

A product of goal refinement is development of a set of *benefit functions* for each product family. These are simple mathematical models of the significance of each feature of the product family, built before definition of a platform architecture or its implementation. The product family architecture team can further decompose benefit functions into characteristic functions, through a process of iterative refinement with the product family stakeholders.

We have identified some generic goals for SOC development, summarized as benefit functions in Figure 4:

- Reduce the complexity of product development tasks
- Minimize time to market (TTM) through reuse
  - Minimize SOC time to market
    - TTM of first product in the product family
    - TBSP (time between successive products) for derivatives/variants
  - Maximize reuse of basic assets
- Partition the verification of an SOC into manageable pieces
- Leverage existing RTOS ports and third-party development tools
- Offer customers consistency in programming models

Example:
Objective #1: Reduce the <u>complexity</u> of <u>development tasks</u>

Questions:
- How is *complexity* defined?
  - The number of total process steps required to develop and deliver all the members of a given product family over its lifetime
- What are the *development tasks*? (Using the product family approach)
  - Product family business plan and opportunities worksheet development and maintenance
  - Product family requirements specification and roadmap alignment (approved by both customer and ASP core development team)
  - Product family/platform architecture specification
  - Development, verification, evolution, and maintenance of platform
  - Development/acquisition, verification, and maintenance of variable/differentiating product specific IP
  - Development/acquisition, verification, and maintenance of collateral IP
  - Development, verification and delivery of each member of the product family according to the New Product Introduction (NPI) key stages

Benefit Functions:

Difference between number of process steps required for development of a set of products using the product family approach and an equal number developed in the traditional one-at-a-time approach

$$N_{pf} = N_{pfplan} + N_{pfpd} + N_{pfcoll} + \Sigma_p((N_{pplan}(p) + N_{ipdev}(p) + N_{plinteg}(p) + N_{pver}(p))$$

$$N_{trad} = \Sigma_p((N_{pplan}(p) + N_{pdev}(p) + N_{pinteg}(p) + N_{ver}(p) + N_{poll}(p))$$

Where:

- N is the number of process steps for each activity
- p = per-product index
- pf = product family approach
- trad = traditional approach

Result:

Benefit from reduced complexity accrues when

$N_{pf} < N_{trad}$, thus $\mathbf{B = N_{trad} - N_{pf}}$

Figure 4: Benefit Functions

Another significant advantage of the product family approach is the ability to comprehend an entire product family development as part of a single technical, marketing, and financial analysis. Product family economic analysis addresses investment and return information for both revenue producing product family members and required, but non-revenue producing items, such as platforms themselves, reusable differentiating IP, collateral, tools, and support (see "Platform Approach Requires More" below). Initial costs for platform, IP, and collateral development that were previously borne by a single platform product–and which thus became barriers to effective platform-based design—can be more easily amortized across all the product family members, thus improving net margins for the entire family.

Finally, the product family approach is completely scaleable, and applicable at any level of integration platform. For example, the same process ASP engineers use to define wireless domain core platform architectures can be adopted by WBSG engineers developing cellular merchant market SOC platforms, and by handset designers in response to feature set needs expressed by network service providers and their customers, the end users.

Some of these collateral components will be addressed by the OPEN project, including integration with the emerging SPS system-level design flow, and enhancement of our New Product Introduction (NPI) process Opportunity Selection Worksheet (OSW) [9, 10] to look at economic models of cost, return, and payback for an entire product family.

Architectural Description Standards

The functional and collateral components of core platforms form the basis for formal architectural descriptions we'll develop. Architectural descriptions contain information about the platform from the perspective of all its stakeholders, identifying all their concerns, and addressing those concerns via several architectural viewpoints. In the same way a set of blueprints represents a building with a variety of separate specifications of its structural, electrical, plumbing, and other attributes, a platform architectural description will include views describing all of its many relevant attributes. We will use the recently published **IEEE Recommended Practice for Architectural Description of Software Intensive Systems** [6] as a style guide.

We expect a variety of viewpoints to be represented in our core platform architectural descriptions, each supported by appropriately instantiated models and supported evaluation practices, including these views:

Table 2: Platform Architectural Views

• Behavioral, dynamic, operational	• Data structure, data flow, information
• Development, maintenance	• Test, validation & verification (DFT)
• Manufacturing (DFM)	• Distributed, network, multiprocessor
• Functional, activity	• Logical
• Integration flow	• Integration by acquirer into higher level system
• Reusability and the SRS (DFR)	• Static, structural
• Physical, interface, interconnect	• Reliability, quality, availability, safety

Platform Approach Requires More

Customers' design decisions are increasingly made on what's best supported and easiest for them to take to market. In addition to the platforms and functional components described above, we believe platforms must be delivered with several collateral components to make them "useful, desirable, and convenient" [12], including development and integration tools; support practices; and documentation and software.

Development and integration tools include a full, integration-oriented, platform-based embedded system design flow, tool set, and verification methodology, along with application and system engineering support for hardware and software designers and integrators using the platform. Reference designs and implementations in the platform's application domain are crucial. *Support practices* cover both the platform developer, and the integrator-customer, including platform architecture and development system training, and bug reporting and change management facilities. Also important to both integrators and platform developers are platform roadmaps and life cycle plans (evolution, extension, migration, end-of-life, etc.), and economic models of product families. Finally, *documentation and software* used to assist in design integration, physical integration, and verification or test of a platform includes a platform architectural description; instruction set simulators; system-level models (behavioral, functional, simulation, test, formal verification, etc.); platform characterization (performance, power, size, etc.); linkage to standards references (development, operation, interface, etc.).

Summary

ASP is embarking on our platform engineering methodology program, a new approach to product development based on platform based design, that recognizes the increasing complexity of the embedded systems our customers want to build, and the increasingly competitive nature of the markets in which both we and they compete. We're ready to set new goals for SPS to perform to and succeed in each of these areas:

- A common, reusable approach to platform architecture derivation, applicable a all levels of integration
- A well defined set of product families, and roadmaps to enhance them over time
- Vastly improved time to market for our customers and return on our investment, when compared to traditional one-at-a-time product development
- Technology leadership and structured reuse
- Ability to leverage value in IP that isn't tied to a single specific product, but is reusable across several product families: greater opportunity, lower cost
- Better performance at the software-intensive embedded system level
- Commitment to making core and SOC platforms work in our customers' products

References

[1] Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew McNelly, and Lee Todd, **Surviving the SOC Revolution: A Guide To Platform-Based Design**, Boston, MA, Kluwer Academic Publishers, 1999 (http://www1.fatbrain.com/asp/bookinfo/bookinfo.asp?theisbn=0792386795&vm=)

[2] VSIA Platform-Based Design Study Group discussion, January, 2002 (http://vsi.org)

[3] Jean-Marc DeBaud and Klaus Schmid, **A Systematic Approach to Derive the Scope of Software Product Lines**, in Proceedings of the 1999 International Conference on Software Engineering, ACM, pp. 34-49 (http://compass.mot.com/doc/101132423/A_Systematic_Approach_to_Derive_the_Scope_of_Software_Product_Lines.pdf)

[4] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K Schmid, T. Widen, and J.-M. DeBaud, **PuLSE: A Methodology to Develop Software Product Lines**, in Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR99), Los Angeles, CA, May, 1999, pp. 122-131

[5] Victor Basili, Gianluigi Caldiera, and Dieter Rombach, **The Goal-Question-Metric Paradigm**, Encyclopedia of Software Engineering, pp 528-532, John Wiley & Sons, 1994

[6] IEEE Std. 1471-2000, **IEEE Recommended Practice for Architectural Description of Software Intensive Systems** (http://compass.mot.com/go/ieee1471)

[7] William H. Davidow, **Marketing High Technology: An Insider's View**, Free Press, 1986 (http://www1.fatbrain.com/asp/bookinfo/bookinfo.asp?theisbn=002907990X&vm=)